苏 州 大 学 实 验 报 告

院、系	计算机学院	年级专业	软件工程	姓名	朱金涛	学号	2327406014
课程名称	操作系统课程实践					成绩	
指导教师	王红玲	同组实验者	无		实验日期	2025 年 11 月 24 日	

实 验 名 称　字符设备驱动程序编写

一、 实验目的

（1）了解 Linux 字符设备管理机制。

（2）学习字符设备的基本管理方法。

（3）学会编写简单的字符设备驱动程序的方法。

二、 实验内容

编写一个字符设备驱动程序，要求能对该字符设备执行打开、读、写、I/O 控制和关闭这 5 项基本操作。并编写测试程序，测试添加的字符设备的正确性。

三、 实验步骤和结果

本实验旨在编写一个 Linux 字符设备驱动程序，实现内核空间与用户空间的数据交互。实验流程分为**复现书中源码**中的基础功能的实现（设备注册、打开、读取）以及**“写”功能的自我实现。**

1. 驱动编译与读取测试

步骤 1：编译内核模块与用户测试程序 首先清理旧的编译文件，确保环境纯净。使用 make 命令调用内核构建系统编译 chardev.c 生成内核模块 .ko 文件，使用 gcc 编译用户测试程序 test.c。

- 操作命令：

```
make

gcc test.c -o test
```

问题解决：

当直接使用书本上的 **chardev.c** 进行 **make** 时，终端会有 **warning**：

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ make
make -C /lib/modules/6.17.014Jintao/build M=/home/tt/下载/2327406014_朱金涛_实验
九_源码 modules
make[1]: 进入目录“/home/tt/下载/linux-6.17.2”
make[2]: 进入目录“/home/tt/下载/2327406014_朱金涛_实验九_源码”
  CC [M]  chardev.o
chardev.c:67:5: warning: no previous prototype for ‘init_chardev_module’ [-Wmiss
ing-prototypes]
   67 | int init_chardev_module(void)
      |     ^~~~~~~~~~~~~~~~~~~
chardev.c:85:6: warning: no previous prototype for ‘exit_chardev_module’ [-Wmiss
ing-prototypes]
   85 | void exit_chardev_module(void)
      |      ^~~~~~~~~~~~~~~~~~~
chardev.c: In function ‘device_read’:
chardev.c:55:5: warning: ignoring return value of ‘copy_to_user’ declared with a
ttribute ‘warn_unused_result’ [-Wunused-result]
   55 |     copy_to_user(buffer, msg_Ptr, length);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  MODPOST Module.symvers
WARNING: modpost: missing MODULE_DESCRIPTION() in chardev.o
  CC [M]  chardev.mod.o
  CC [M]  .module-common.o
```

主要原因在于代码安全性检查缺失与编码规范陈旧，不符合现代 Linux 内核的严格要求。

具体表现为：代码**忽略了** copy_to_user 函数的返回值（该函数带有__must_check 属性，忽略它会导致数据传输失败时不报错的安全隐患），且模块的初始化与退出函数**未声明**为 static（导致编译器提示全局函数缺少原型），同时**缺少** MODULE_DESCRIPTION 描述信息。

解决方法是：在代码中增加了对 copy_to_user 返回值的判断(若非零则返回 -EFAULT)，将 init 和 exit 函数加上 static 关键字以限制其作用域，并补全了缺失的模块描述宏，从而完

全消除了编译警告并提升了驱动的健壮性。

```c
static int is_user_msg = 0; // 新增：标记当前消息是否由用户写入

// 函数原型声明
static int device_open(struct inode *inode, struct file *file);
static int device_release(struct inode *inode, struct file *file);
static ssize_t device_read(struct file *filp, char __user *buffer, size_t length, loff_t *offset);
static ssize_t device_write(struct file *filp, const char __user *buff, size_t length, loff_t *off);

static struct file_operations fops = {
    .read = device_read,
    .write = device_write, // 注册写函数
    .open = device_open,
    .release = device_release};

// 打开设备
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (Device_Open)
        return -EBUSY;

    Device_Open++;

    // 【关键修改】只有当用户没有写入过自定义消息时，才使用默认消息
    if (!is_user_msg)
    {
        sprintf(msg, "I already told you %d times Hello world\n", counter++);
    }

    msg_Ptr = msg; // 重置读指针到开头
    try_module_get(THIS_MODULE);
    return SUCCESS;
}
```

改进后：

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ make
make -C /lib/modules/6.17.014Jintao/build M=/home/tt/下载/2327406014_朱金涛_实验九_源码 modules
make[1]: 进入目录“/home/tt/下载/linux-6.17.2”
make[2]: 进入目录“/home/tt/下载/2327406014_朱金涛_实验九_源码”
  CC [M]  chardev.o
  MODPOST Module.symvers
  CC [M]  chardev.mod.o
  CC [M]  .module-common.o
  LD [M]  chardev.ko
  BTF [M] chardev.ko
make[2]: 离开目录“/home/tt/下载/2327406014_朱金涛_实验九_源码”
make[1]: 离开目录“/home/tt/下载/linux-6.17.2”
```

成功编译且无警告

步骤 2：加载内核模块 使用 insmod 命令将编译好的驱动模块动态加载到 Linux 内核中。

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ sudo insmod
chardev.ko
[sudo] tt 的密码：
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ lsmod | grep
 chardev
chardev                16384  0
```

成功加载内核模块

步骤 3：查看系统日志分配的主设备号 驱动程序在注册时动态申请了主设备号，我们需要通过内核环形缓冲区日志（dmesg）查看系统实际分配的号码，以便后续创建设备节点。

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ sudo dmesg
 tail
[ 3662.736038] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 7 times, c
nsider switching to WQ_UNBOUND
[ 3714.396932] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 11 times,
onsider switching to WQ_UNBOUND
[ 4243.417277] workqueue: psi_avgs_work hogged CPU for >10000us 11 times, consi
er switching to WQ_UNBOUND
[ 4587.185331] workqueue: blk_mq_run_work_fn hogged CPU for >10000us 19 times,
onsider switching to WQ_UNBOUND
[ 5207.151031] chardev: loading out-of-tree module taints kernel.
[ 5207.151497] chardev: module verification failed: signature and/or required k
y missing - tainting kernel
[ 5207.172348] I was assigned major number 239
[ 5207.172354] To talk to the driver, create a dev file with:
[ 5207.172358] mknod /dev/chardev c 239 0
[ 5207.172362] Remove the device file and module when done
```

结果分析：日志显示驱动初始化函数 init_chardev_module 被调用，并打印出了分配的主设备号（239），这验证了 module_init 宏的作用。

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ sudo mknod /
dev/hello c 239 0
```

mknod 是 Linux 系统中手动创建一个“设备节点文件”（/dev/chardev）。当用户读写

这个文件时，操作系统会根据主设备号（239），自动转接到驱动程序上。用来**连接驱动程序与 test.c**

Test.c 文件核心代码如下：

```c
// --- 测试默认读取 ---
printf("1. 打开设备读取默认消息...\n");
fd = open(DEVICE_FILE, O_RDWR);
if (fd < 0)
{
    perror("打开失败");
    return -1;
}

ret = read(fd, read_buf, sizeof(read_buf) - 1);
if (ret > 0)
{
    read_buf[ret] = '\0';
    printf("-> [Read Default]: %s", read_buf);
}
close(fd); // 关闭设备
```

步骤 5：运行测试程序验证读取功能 运行./test 程序。该程序会打开设备文件，从内核读取数据并打印。多次运行以观察驱动内部计数器的变化。

问题解决：

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
I have already told you 1 time hello world!
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
I have already told you 1 time hello world!
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
I have already told you 1 time hello world!
```

直接用了书上的源码，但是运行发现它**将输出语句写死了**，不管怎么调用都只会输出“...told you 1 time…”。

```c
// 打开设备
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (Device_Open)
        return -EBUSY;

    Device_Open++;

    // 【关键修改】只有当用户没有写入过自定义消息时，才使用默认消息
    if (!is_user_msg)
    {
        sprintf(msg, "I already told you %d times Hello world\n", counter++);
    }

    msg_Ptr = msg; // 重置读指针到开头
    try_module_get(THIS_MODULE);
    return SUCCESS;
}
```

于是我**重写代码**，将缓冲区初始化为空，确保 open 函数打开的设备节点名称（/dev/chardev）与驱动注册的一致，并增加对 read 返回值的检查，强制程序打印从内核实际读取到的数据；同时检查驱动代码，确保计数器变量声明为 static 以维持状态。重新编译并加载模块后，程序成功输出了由内核动态生成的、数值递增的交互信息。

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
读到的内容：I already told you 4 times Hello world
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
读到的内容：I already told you 5 times Hello world
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
读到的内容：I already told you 6 times Hello world
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
读到的内容：I already told you 7 times Hello world
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
读到的内容：I already told you 8 times Hello world
```

结果分析：

- 程序成功打开设备，输出了从内核读取的字符串。
- 输出内容依次为 ... 4 times ..., ... 5 times ..., ... 6 times ...。
- 这证明了驱动中的 device_open 函数被正确调用，且内部的 static int counter 变量**成功在内存中驻留**并累加，未随函数结束而销毁。

2. “写”功能的实现

为了验证双向数据传输，我在**原实验基础上修改**了驱动代码，增加了 device_write 函数，支持用户向内核写入自定义消息。

```c
// 【新增功能】写设备
static ssize_t device_write(struct file *filp, const char __user *buff, size_t length, loff_t *off)
{
    int copy_len = length;

    // 1. 防止缓冲区溢出 (保留1个字节给结束符 \0)
    if (copy_len > BUF_LEN - 1)
    {
        copy_len = BUF_LEN - 1;
    }

    // 2. 从用户空间拷贝数据到内核 msg 数组
    // copy_from_user 返回未拷贝成功的字节数，成功返回0
    if (copy_from_user(msg, buff, copy_len))
    {
        return -EFAULT;
    }

    // 3. 必须手动添加字符串结束符，因为用户传来的未必包含 \0
    msg[copy_len] = '\0';

    // 4. 标记为“用户自定义消息”，这样下次 open 就不会覆盖它了
    is_user_msg = 1;

    // 5. 更新读指针（可选，重置指针以便立即可以读取刚才写入的内容）
    msg_Ptr = msg;

    pr_info("Driver: User wrote %d bytes: %s\n", copy_len, msg);

    // 返回实际写入的字节数
    return copy_len;
}
```

步骤 6：修改驱动逻辑与重新部署 在 chardev.c 中添加了 copy_from_user 函数以接收用户数据，并增加逻辑判断：若用户写入了数据，读取时不再覆盖为默认的 Hello World 消息。修改完成后重新编译并加载。

编写测试文件 test.c 并对结果进行测试：

关于“写”功能测试的 test.c 部分代码：

```c
// --- 第二步：测试写入功能 ---
printf("\n2. 打开设备并写入新消息...\n");
fd = open(DEVICE_FILE, O_RDWR);
if (fd < 0)
{
    perror("打开失败");
    return -1;
}

printf("-> 正在写入: \"%s\"\n", write_buf);
ret = write(fd, write_buf, strlen(write_buf));
if (ret < 0)
{
    perror("写入失败");
}
else
{
    printf("-> 写入成功! 写入了 %d 字节\n", re
});
close(fd);

// --- 第三步：验证写入结果 ---
printf("\n3. 再次打开设备验证写入内容...\n");
fd = open(DEVICE_FILE, O_RDWR);
if (fd < 0)
{
    perror("打开失败");
    return -1;
}

memset(read_buf, 0, sizeof(read_buf));
ret = read(fd, read_buf, sizeof(read_buf) - 1);
if (ret > 0)
{
    read_buf[ret] = '\0';
    printf("-> [Read New]: %s\n", read_buf);
}
close(fd);
```

运行查看结果：

```
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
1. 打开设备读取默认消息...
-> [Read Default]: I already told you 0 times Hello world

2. 打开设备并写入新消息...
-> 正在写入: "This is a NEW message from User!"
-> 写入成功! 写入了 32 字节

3. 再次打开设备验证写入内容...
-> [Read New]: This is a NEW message from User!
tt@tt-VMware-Virtual-Platform:~/下载/2327406014_朱金涛_实验九_源码$ ./test
1. 打开设备读取默认消息...
-> [Read Default]: This is a NEW message from User!
2. 打开设备并写入新消息...
-> 正在写入: "This is a NEW message from User!"
-> 写入成功! 写入了 32 字节

3. 再次打开设备验证写入内容...
-> [Read New]: This is a NEW message from User!
```

结果分析：

- **Read Default:** 第一次读取显示默认消息。
- **Writing:** 提示写入成功，驱动日志（dmesg）中也应记录了写入的字节数。
- **Read New:** 再次读取时，内容变为用户写入的 "This is a NEW message..."。
- 这证明数据成功从 **用户空间 -> 内核空间（存储在 msg 数组） -> 用户空间** 完成了完整的闭环流转。

四、 实验总结

本次实验通过编写和测试 Linux 字符设备驱动，深入理解了 Linux 操作系统中设备驱动的工作原理及内核模块的开发流程。

1. 核心知识点回顾：

- **内核空间与用户空间隔离：** 实验深刻体现了两者不能直接进行内存访问的特性。数据的交互必须通过内核提供的安全接口 copy_to_user（读）和 copy_from_user（写）

来完成，这保证了内核的稳定性。

- **设备文件机制：** 理解了 VFS（虚拟文件系统）如何通过主设备号将 /dev 下的设备文件映射到具体的驱动程序结构体 file_operations 上。
- **模块生命周期：** 掌握了 module_init 和 module_exit 宏的作用，以及如何使用 insmod 和 rmmod 管理内核模块。

2. 遇到的问题与解决方案：

- **问题一：编译警告。** 在初次编译时，出现了忽略 copy_to_user 返回值的 Warning。
  - *解决：* 查阅文档得知该函数返回未成功拷贝的字节数。我在代码中增加了对返回值的判断，若非 0 则返回-EFAULT，增强了代码的健壮性。
- **问题二：测试程序逻辑误区。** 最初测试时，test.c 中直接打印了硬编码的字符串，导致无法判断是否真正读取了驱动数据。
  - *解决：* 修改测试代码，检查 read 函数的返回值，并打印缓冲区 buf 中的实际内容，从而通过输出的动态变化（0 times, 1 times...）验证了驱动逻辑。
- **问题三：并发与状态保存。** 实验中观察到计数器 counter 能够累加。
  - *思考：* 这得益于在该变量声明时使用了 static 关键字，使其存储在全局数据区而非栈上。这提示在驱动开发中，需要根据数据是否需要跨调用保存来合理选择存储类别。

3. 思考：

在拓展“写”功能时，我发现如果不加标志位控制，每次 open 设备时 sprintf 会覆盖之前写入的数据。通过引入 is_user_msg 标志位，成功实现了“用户写入优先”的逻辑。这让我

意识到驱动程序不仅是数据的搬运工，更是设备状态的管理者。

通过本次实验，我不仅掌握了字符设备驱动的框架搭建，更在调试过程中提升了对 Linux 内核机制的理解，为后续进行更复杂的块设备或网络设备驱动开发打下了基础。